Carping About DBI

By Vikram Vaswani

Table of Contents

Free−For−All

One of the nice things about Perl − in fact, the only thing about it that makes wading through all those strange hashes, arrays and regexes worthwhile − is the huge amount of free code Perl developers have access to.

These code "modules", or standalone chunks of Perl code, are usually expressly designed for a specific purpose, and are therefore both focussed and efficient. For example, you might need a piece of Perl code to convert HTML to plain text, or a widget to calculate the value of pi to the 67th decimal place. These and other Perl modules are all available for fre, online − if you know where to look!

Over the next few pages, I'm going to take you on a brief tour of two of the most popular and useful Perl modules out there − DBI and Carp. Read on!

Dissecting The DBI

I'll begin with the DBI, a database−independent software interface for Perl. As the name suggests, it's a layer of abstraction over the actual database access methods and allows developers to deal with different databases without radically altering their code on a per−database basis. As long as a "DBI drive" (aka DBD) exists for the database in question, you're home free!

By placing a layer of abstraction between the database and the developer, the DBI insulates the programmer from database implementation details. If you initially write a script to talk directly to, say, Oracle and later need to have it work with another database server, you will usually have to rewrite all the database−specific parts. If you use a database API like the DBI, you can port your script over with very little surgery required.

The DBI by itself is not responsible for the interaction between the database and a Perl script. It simply provides a consistent interface to the underlying DBD (which does the actual work). It is the DBD which actually "talks" to the database, not the DBI. When a function call is made from within DBI, it actually transmits the properly−formatted information over to the relevant DBD. Once the DBD is through, it picks up the response and returns it to the caller.

The DBI is also responsible for maintaining a clean interface, handling the automatic loading of DBDs, error checking, and so on. Acting like a middleman, it deals with everything specific to a particular database, thereby allowing the underlying implementation to change without the developer having to worry about it.

There exist DBDs for most popular databases, including MySQL, PostgreSQL, Oracle and others. There's even an ODBC DBD, which can be used to connect to Microsoft databases like Access.

Animal Antics

The best way to understand a new API (and that's all that the DBI is, a consistent database API) is by trying out some sample code.

Before I begin, though, you should make sure that you have the DBI installed on your system. A simple way to test for its presence is to use "perldoc" to look up its documentation.

```
$ perldoc DBI
```

If you can read the documentation, it's a good bet that the module is installed. If it isn't there, then you'll have to download it from http://www.cpan.org/ and install it (note that you'll have to install both the base DBI module and the DBD of whichever database you're using). For the latter part of this article, you might also like to install the Carp module (although it's almost always present) and the CGI module (if you plan to use CGI::Carp).

With that out of the way, here's a simple example which demonstrates some of the functionality of the DBI. Consider the following database table,

```
mysql> SELECT * FROM pets;
+--------+-----------+-----+
| name | species | age |
+--------+-----------+-----+
| Dawg | dog | 5 |
| Rollo | rhinoceros | 7 |
| Polly | parrot | 1 |
| Chucky | chicken | 2 |
+--------+-----------+-----+
4 rows in set (0.00 sec)
```

and then consider this short Perl script, which connects to the database and prints out the data within the table.

```
#!/usr/bin/perl

# load module
```

```
use DBI();

# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658", {'RaiseError' => 1});

# execute query
my $sth = $dbh->prepare("SELECT * FROM pets");
$sth->execute();

# iterate through resultset
while(my $ref = $sth->fetchrow_hashref())
{
print "Name: $ref->{'name'}\nSpecies: $ref->{'species'}\nAge:
$ref->{'age'}\n\n";
}

# clean up
$dbh->disconnect();
```

Here's the output.

```
Name: Dawg
Species: dog
Age: 5

Name: Rollo
Species: rhinoceros
Age: 7

Name: Polly
Species: parrot
Age: 1

Name: Chucky
Species: chicken
Age: 2
```

The script starts off simply enough; as you probably already know, the first line calls the Perl interpreter and tells it to parse and run the statements that follow.

```
use DBI()
```

is the first of those statements. It loads and activates the interface, making it possible to now use the DBI from within the script.

The next line calls the function connect() from within the DBI, and passes it a large number of parameters, including the name of the DBD to use (mysql), the name of the database (somedb), the address of the database server (localhost), and the database username and password.

Opening a connection to the database is generally an expensive operation, requiring a certain amount of time and consuming a certain amount of system resources. It's best to connect just once at the beginning, and then disconnect at the end.

Next, a call to RaiseError ensures that if the DBI encounters an error, it will die() rather than return an error value. For a simple script like the one above, this works out pretty well; however, in more complicated scripts, you might prefer to turn this off and handle errors in a more intelligent manner.

Note that there is no standard for the string that follows a DBD name; it differs in format from DBD to DBD. You'll need to consult the documentation that came with your DBD to obtain the format specific to your database.

As you can see, connect() returns a handle to the database, which is used for all subsequent database operations. This also means that you can open up connections to several databases simultaneously, using different connect() statements, and store the returned handles in different variables. This is useful if you need to access several databases at the same time; it's also useful if you just want to be a smart−ass and irritate the database administrators. Watch them run as the databases over heat! Watch them scurry as their disks begin to thrash! Watch them gibber and scream as they melt down!

The prepare() function, which takes an SQL query as parameter, readies a query for execution, but does not execute it (kinda like the priest that walks down the last mile with you to the electric chair). Instead, prepare() returns a handle to the prepared query, which is stored and then passed to the execute() method, which actually executes the query (bzzzt!).

Although overkill for our simple needs, you should be aware that prepare() can provide a substantial performance boost in certain situations. Many database scripts involve preparing a single query (an INSERT, for example) and then executing it again and again with different values, and using a prepare() statement in such a situation can help reduce overhead.

Once the query has been executed, the next order of business is to do something with the returned data. A number of methods are available to iterate through the resultset and parse it into different fields − I've used the fetchrow_hashref() method to pull in the data as hash references and format it for display. I could also have printed out the entire table line by line using fetchrow_array() − this will be demonstrated in the next example.

Once all the data has been retrieved, the disconnect() function takes care of disengaging from the database (freeing up memory and generally cleaning things up).

Do()ing More

Of course, there's a lot more to the DBI than what you've just seen. For example, if you don't like the thought of prepare()−ing and execute()−ing a query, you can take a shortcut with the do() method, used to execute a query directly.

```
#!/usr/bin/perl

# activate module
use DBI();

# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658");

# execute query
my $rows = $dbh->do("UPDATE pets SET age = age+1 WHERE name =
'polly'");
print "$rows row(s) affected\n";

# clean up
$dbh->disconnect();
```

It makes more sense to use do() for SQL statements like CREATE and DROP than to laboriously prepare() and then execute() them. However, if you're going to be running a similar query over and over again, prepare() and execute() will be more efficient. Additionally, do() can't be used for SELECT queries because a statement handle is needed for further data extraction, and do() only returns the number of rows affected by the query (or an undef on error), making it impossible to use for these kinds of operations.

The rows() method can be used to discover the number of rows affected by the last SQL query. However, don't try to use this after an SQL SELECT query (it's better for INSERTs and DELETEs), because the only way most drivers can figure out the number of rows affected is to actually count them as they are returned. Here's an example:

```
#!/usr/bin/perl

# activate module
use DBI();
```

```
# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658", {'RaiseError' => 1});

# execute query
my $sth = $dbh->prepare("UPDATE pets SET age=age+1 WHERE
name='polly'");
$sth->execute();

# print affected rows
print $sth->rows() . " row(s) affected\n";

# clean up
$dbh->disconnect();
```

If you need special characters quoted correctly, quote() will leap to your rescue.

```
#!/usr/bin/perl

# activate module
use DBI();

# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658", {'RaiseError' => 1});

# execute query
$name = $dbh->quote("Godzilla's");
$dbh->do("INSERT INTO pets VALUES ($name, 'Iguana', 4)");

# clean up
$dbh->disconnect();
```

Finally, the finish() method is used to indicate that no more rows will be processed from the resultset. This allows the database to clean up some of the buffers at its end.

```
#!/usr/bin/perl
```

```
# activate module
use DBI();

# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658", {'RaiseError' => 1});

# execute query
my $sth = $dbh->prepare("SELECT * FROM pets");

// snip

# clean up
$sth->finish();
$dbh->disconnect();
```

This function is not really required, but it's a good idea to use it if you can. It keeps your code clean, and makes other programmers think you know more than you do, which is always useful.

When Things Go Wrong

No matter how good you are, you're bound to screw up sometime. And when it happens, you'll be glad that I told you about $DBI::errstr, used to display error messages when things go wrong.

```
# connect
my $dbh =
DBI->connect("DBI:mysql:database=somedb;host=localhost", "me",
"me545658") || die "Can't connect: $DBI::errstr";
```

Additionally, you can also check the handle for errors during a query − as the following example demonstrates.

```
while(my $ref = $sth->fetchrow_hashref())
{
print "Name: $ref->{'Name'} Species: $ref->{'Species'} Age:
$ref->{'Age'}
\n";

if($sth->err)
{
die "There was an error: $sth->errstr";
}
```

Speed Demon

The single greatest performance boost you can realize with DBI, is through the intelligent use of prepare() and execute() functions. Whenever you find yourself using a particular SQL statement over and over again, prepare() it and call it using execute().

However, it rarely happens that the very same query, with no difference at all, is executed over and over again. For example, suppose you're trying to INSERT some data into the database. The INSERT statement may remain similar in form, but the data values will keep changing every time. In this case, prepare() and execute() must be used in combination with DBI "placeholders", as demonstrated below.

```
$sth = $dbh->prepare("INSERT INTO pets VALUES (?,?,?)");
$sth->execute($name, $species, $age);
```

The questions marks you see in the prepare() statement are placeholders, and they function just like variables; they indicate where the data values for the current query will be placed later. The data values themselves come from the subsequent execute() statement. Typically, the execute() statement is placed in a loop and the variables $name, $species and $age change every time. The data could be entered interactively at the command prompt, read in from a file, or injected intravenously.

It's also important to be careful about the number of open database connections while using DBI. All databases have a finite limit on the number of concurrent client connections they can handle. If the maximum number of concurrent clients is unspecified, it may make more sense to open up a new connection only when required and to shut it down as soon as possible. This is necessary to ensure that there is never a time when the database server is so heavily loaded that it cannot accept any new connections and is forced to turn away client programs.

Unfortunately, opening and closing connections is computationally expensive and time-consuming. You'll have to find a balance between keeping connections open, and loading the server. This is something that you can't apply in a cookbook fashion; it depends on many variables and the final compromise will differ from implementation to implementation.

Dummy Data

Those of you who've been programming for some time know the value of error messages − they're a great way to scare the pants off clueless users. They also have other, more mundane uses, like actually helping developers track down bugs in their scripts (yeah, right!) and gracefully handling errors as and when they occur.

Perl comes with two very important and useful error−handling functions, die() and warn(). You're probably already familiar with both of these; die() is used to kill a rogue script, printing out an optional status message, while warn() is used to warn the user about possible error conditions by printing out either a user−supplied string or a default message.

Unfortunately, sometimes more descriptive errors are needed. Although die() and warn() work great for printing out errors and warnings, they don't offer this information from the perspective of the "caller" in the program. For example, when you call die() from within a function, die() will report the error from the location where it is called and not from the perspective of the called function. An example should make this clearer.

```
#!/usr/bin/perl

sub readFile
{
my $filename = shift(@_);
open(FILE, $filename) or die("Cannot locate file!");
print <FILE>;
close FILE;
}

readFile("dummy.txt");
```

If the file "dummy.txt" cannot be found, Perl will exit the script with the following error message:

```
Cannot locate file! at ./carpdemo.pl line 8.
```

What is required, therefore, is something that not only prints out the error when it happens, but also provides information on the sequence of function calls leading up to the error. Something, in short, like the Carp module.

Croak!

The Carp module comes with four functions which are quite symmetrical in form and function. Two functions, carp() and cluck(), only print out warnings, while two other functions, croak() and confess(), terminate the script and also print out a sequence of error messages. Both cluck() and confess() can also print out the stack backtrace, making it easier to pinpoint the problem.

The Carp module can be used wherever descriptive error messages warning you about potential problems are required. They're not going to make your program run any better, but they will simplify and streamline the process of error detection and recovery.

It should be noted that although, they provide much−needed information, all the Carp functions internally call warn() or die(). However, since they're very lightweight and offer numerous advantages, I'd still recommend that you use them in preference to both die() and warn().

I've rewritten the example above to demonstrate the utility of Carp.

```
#!/usr/bin/perl

use Carp;

sub readFile
{
my $filename = shift(@_);
open(FILE, $filename) or croak("Cannot locate file!");
print <FILE>;
close FILE;
}

readFile("dummy.txt");
```

And here's the output.

```
Cannot locate file! at ./carpdemo.pl line 8
main::readFile('dummy.txt') called at ./carpdemo.pl line 13
```

Whining Some More

The Carp module also comes with a bunch of additional features designed specifically for use in Perl CGI scripts. These features make it easier to track errors thrown up by your CGI scripts, and to log these errors appropriately.

When running as a CGI script, your error output is automatically redirected to the HTTP server error log. Unfortunately, the output is not time−stamped, making it difficult to figure out when the error occurred. To get time−stamped error and warning messages, simply replace the line

```
use Carp;
```

with

```
use CGI::Carp;
```

The rest of your code remains the same.

Another nifty little feature here is the ability to echo error messages (though not warnings) to the client browser. Simply import the fatalsToBrowser() subroutine and this will be done for you automagically. Change the line

```
use CGI::Carp;
```

to

```
use CGI::Carp qw(fatalsToBrowser);
```

Now, not only will errors be recorded in the log, but they will also be displayed to the user. This may or may not be something you want (do you really want the world to see the error messages your shoddy little script vomits out?), which is why it has to be turned on manually by importing fatalsToBrowser().

Finally, the carpout() function allows you to redirect error messages to a file of your choosing, rather than sending them to the standard error log. This can be a great help while debugging, since you won't have to wade through miles of log file messages in order to find the hairballs you script coughed up.

Here's an example:

```
BEGIN {
use CGI::Carp qw(carpout);
open (CARPLOG, ">>/usr/home/luser/my-script.log") or die
("Can't open log
file!");
carpout(CARPLOG);
}
```

As you can see, carpout() has to be specially imported in before it can be used. carpout() needs just one parameter, the file handle of the user-defined error log.

Keep in mind that carpout() degrades performance and should not be used in an actual production script. Use it only as a tool for debugging your script, or as an excuse to explain why it's so slow.

Final Thoughts

Both DBI and Carp are extremely useful modules. DBI is being worked on extensively and is very popular amongst Perl coders. Its popularity ensures that database drivers are written for even the most obscure platforms and then kept up to date. It's not too difficult to write your own database drivers, so if you have some really strange database you have to interface with, you can put in a little more work and still use DBI.

DBI also makes the process of porting code between databases trivial. If your code is general enough and doesn't use any database specific tricks, then all you may have to do is change the DBD string in the connect() function. Light, fast, well−designed and very well−supported, this is a gem of a module.

Carp is also pretty popular. Many developers still prefer to use die() and warn() directly, but once you're hooked onto Carp, it's difficult to let go. It's a very small module, and using it entails a negligible performance and size hit. It's also small enough to read and understand...which means that it can be easily modified or customized.

Both these modules come with good documentation, although the documentation which ships with DBI is far more complete. Be sure to not only read the DBI documentation, but also the documentation for the relevant DBD, since it will usually contain sample code and detailed information about each and every available function.

You can obtain more information on both these modules at any of the links below:

The DBI FAQs at http://theoryx5.uwinnipeg.ca/cgi−bin/doc−filter?page=data/DBI/DBI/FAQ.html;query=dbi;match=and;where=all;stem=;type=data and http://theoryx5.uwinnipeg.ca/cgi−bin/doc−filter?page=data/DBI/DBI.html;query=dbi;match=and;where=all;stem=;type=data

Perldoc documentation for DBI at http://www.perldoc.com/cpan/DBI.html

Perldoc documentation for Carp at http://www.perldoc.com/cpan/CGI/Carp.html

"A Short Guide To DBI" by Mark−Jason Dominus at http://www.perl.com/pub/1999/10/DBI.html

Check 'em out, and I'll see you soon!